




Characterizing Security-Related Commits of JavaScript Engines

Bruno Gonçalves de Oliveira¹^a, Andre Takeshi Endo²^b and Silvia Regina Vergilio¹^c

¹*Department of Computer Science, Federal University of Paraná, Curitiba, PR, Brazil*

²*Computing Department, Federal University of São Carlos, São Carlos, SP, Brazil*

Keywords: Security, JavaScript, Software Vulnerability, Metrics, Mining Repositories.

Abstract: JavaScript engines are security-critical components of Web browsers. Their different features bring challenges for practitioners that intend to detect and remove vulnerabilities. As these JavaScript engines are open-source projects, security insights could be drawn by analyzing the changes performed by developers. This paper aims to characterize security-related commits of open-source JavaScript engines. We identified and analyzed commits that involve some security aspects; they were selected from the widely used engines: V8, ChakraCore, JavaScriptCore, and Hermes. We compared the security-related commits with other commits using source code metrics and assessed how security-related commits modify specific modules of JavaScript engines. Finally, we classified a subset of commits and related them to potential vulnerabilities. The results showed that only six out of 44 metrics adopted in the literature are statistically different when comparing security-related commits to the others, for all engines. We also observed what files and, consequently, the modules, are more security-related modified. Certain vulnerabilities are more connected to security-related commits, such as Generic Crash, Type Confusion, Generic Leak, and Out-of-Bounds. The obtained results may help to advance vulnerability prediction and fuzzing of JavaScript engines, augmenting the security of the Internet.


1 INTRODUCTION


JavaScript engines are responsible for compiling and executing code of highly-interactive pages and are essential for modern Web browsers. The engines run locally (in the host computer) JavaScript code served from any web page on the Internet accessed by end users; this makes them a security-critical part of the browser. As a run time compiler with particular assets in memory, the vulnerabilities in the engines could allow hijacking system execution with unique resources manipulated for exploitation. That occurs because JavaScript engines have special features associated with the real-time compiler that may generate distinct vulnerabilities. The engines have different mechanisms for code compilation, so when the same block of JavaScript code is executed more than once, it may trigger in particular cases optimization mechanisms causing flawed outcomes; this brings specific security risks (Kang, 2021). The engines may have particular issues due to memory allocations and improper data validation that turn them promising targets for secu-


rity research.

Due to these characteristics, JavaScript engines have been a trendy topic of many security conferences and projects, e.g., OffensiveCon (OffensiveCon, 2022) and the CodeAlchemist project (Han et al., 2019). Although there is an extensive body of knowledge about software security in general, there is still much to be addressed regarding JavaScript engines security. In addition to industrial initiatives, like National Vulnerability Database (NVD) (US-Government, 2023), there are initiatives from academia to create datasets that extend CVE (Common Vulnerabilities and Exposures) entries from NVD with commits' information and source code metrics (Gkortzis et al., 2018; Kiss and Hodován, 2019). This kind of datasets has been used to support empirical studies on software vulnerabilities and their relation to different metrics (Zaman et al., 2011; Alves et al., 2016; Iannone et al., 2022). In general, the end goal is to come up with metrics that are good predictors for pinpointing unknown vulnerabilities in the source code (Shin and Williams, 2008; Shin et al., 2011; Jimenez et al., 2019).

Furthermore, most of these studies are built on top of labeled data from NVD or other repositories; this

^a <https://orcid.org/0009-0008-4554-5166>

^b <https://orcid.org/0000-0002-8737-1749>

^c <https://orcid.org/0000-0003-3139-6266>

may bias the results in favor of publicly known vulnerabilities and overlook other relevant security aspects. Another point is that most research on vulnerabilities introduces approaches to prevent, detect, and fix issues which are extracted only based on the code (Lin et al., 2019; Park et al., 2020; Mao et al., 2018). But other related software elements can also be considered, e.g. information extracted from commits. The commits are mainly patch codes that change the current application code base for some specific reasons, including security. Any project's commit can be retrieved and analyzed, investigating its timeline, and code before and after the patch.

All these points serve as motivation for the present work, which utilizes commit information to characterize security-related code of open-source JavaScript engines. The main idea is to identify security-related commits by analyzing the content of commits messages. Once such commits are identified, the main characteristics of the modified code are captured through software metrics. The modified files are also related to specific modules of JavaScript engines, and for a subset of these commits, the vulnerability type present in the code is analyzed and classified. In this sense, we go beyond the existing labeled data from NVD and other repositories by selecting any security-related commits. For this end, we selected commits from four widely used engines: V8¹, ChakraCore², JavaScriptCore³, and Hermes⁴.

To identify the security-related commits we used a Machine Learning (ML) classifier. By using the PyDriller and Understand tools, we extracted metrics values for the security-related commits and others non-related ones randomly selected. The results showed statistical difference for only six out of 44 metrics adopted in the literature when comparing security-related commits to the others, for all engines. We also observed that the optimizer and compiler modules of the engines are more security-related modified. Some types of vulnerabilities such as Generic Crash, Type Confusion, Generic Leak and Out-of-Bounds, are the most related to security-related commits in JavaScript engines.

The main contribution of this work is to characterize security-sensitive code and components of JavaScript engines by using information extracted from commits. This represents a novel initiative. The commit texts are generally technical-driven, and secu-

rity context is not always explicit. However, our approach can interpret the commits messages and identifies security-related commits, which bring advantages for our analysis: i) the message description can explain how and why a commit acts on the base code. The messages introduce technical data and social aspects of the commit, allowing readers to understand the context where the commit is applied, consequently to understand the security-related scenarios; ii) many vulnerabilities are being discovered and fixed in JavaScript engines (US-Government, 2023), but some of them are not usually mentioned in the commits, then, a simple search for keywords is not enough; our ML algorithm is capable of finding more relevant commits. This allows the identification of other kinds of vulnerabilities beyond NVD; and iii) the classification permits retrieving an increased number of security-related commits that could be analyzed, creating a bigger dataset to guide future research.

In short, the results herein presented may support future studies in security of JavaScript engines and contributes to a more secure Internet. This study may help to advance vulnerability prediction and fuzzing of JavaScript engines, augmenting the security of Web browsers. To support future replications, we make all the scripts and a detailed guide available as an experimental package⁵.

The paper is organized as follows. Section 2 contains background on JavaScript engines. Section 3 describes the experimental setting. Section 4 presents and discusses the results. Section 5 discuss some threats to validity, while Section 6 reviews related work. Finally, Section 7 concludes the paper.

2 BACKGROUND

The JavaScript language is the key feature to develop rich and dynamic Web applications, complementing HTML and CSS. To run JavaScript code, modern Web browsers such as Google Chrome, Microsoft Edge, Apple Safari, and Mozilla Firefox, have a JavaScript engine in their core. The engine is a piece of software, in those cases written in C/C++, where the JavaScript code is parsed, interpreted, compiled, and optimized. The first JavaScript engine introduced on a browser was the SpiderMonkey, developed for Netscape in 1995. Google released V8, almost ten years later, for Chromium-based browsers in 2014, which is now being used on Microsoft browsers too. Microsoft released the ChakraCore in 2014 as well, to support the

⁵<https://github.com/brunogoliveira-ufpr/security-commits-characterization>

¹<https://v8.dev/>

²<https://blogs.windows.com/msedgedev/2015/12/05/open-source-chakra-core/>

³<https://developer.apple.com/documentation/javascript/core>

⁴<https://reactnative.dev/docs/hermes/>

Microsoft browsers at the time, and then it was replaced in 2018 by V8. Numerous engines have been implemented, some of them target specific contexts. For instance, Hermes is a lean engine optimized for fast start-up of React Native apps, and can be easily integrated to mobile apps.

JavaScript is a multi-paradigm programming language and features dynamic typing. The variables' types are not determined during the compilation; they can only be resolved through dynamic execution (Kang, 2021). The JavaScript engines generally share the same architecture with the following modules: parser, interpreter, baseline compiler, and optimizer, as shown in Figure 1. Initially, the parser generates the Abstract Syntax Tree (AST) from the source code. However, in exceptional cases, if this process takes too long to terminate, the engines might not parse the code entirely; this step operates differently in each engine. The compiler takes the outcome from the interpreter and transforms it into bytecode, along with profiling data. If necessary, the interpreter will send the bytecode and the profiling data to the JIT compiler (optimizer) to speed up the execution.

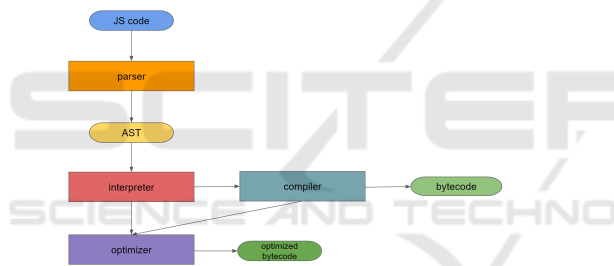


Figure 1: Basic architecture of a JavaScript engine.

The security issues of browsers are mainly related to the JavaScript engines, so the vulnerabilities are usually triggered by exploits written in JavaScript that are loaded into the browser and then coax the end user computer to execute arbitrary commands or other malicious activity. The attacker may access the browser's context while exploiting it, enjoying the user's privilege on the system. There are different vulnerabilities commonly found in JavaScript engines. For example, Use-After-Free (UAF), where a freed memory can be reused, corrupting pre-existing memory allocations. In many cases, allowing the execution of arbitrary remote commands.

The attackers can take advantage of the JavaScript engines vulnerabilities by providing an HTML file with a malicious JavaScript. Figure 2 shows a code snippet that triggers a vulnerability in ChakraCore, identified by CVE-2019-0609 (SSLAB, 2021).

In this case, line 11 defines a `big size object` with enough number of initialized members that will

```

1 function test() {
2   function a() {
3     function d() {
4       let e = function() {};
5       return e;
6     }
7     function b() {
8       let fun_d = [d];
9       return fun_d;
10    }
11    var obj = [big-size object];
12    return b();
13  }
14  return a();
15 }
16 var f = test();
17 function test1() {
18   var obj = [big-size object];
19   print(f[0]);
20 }
21 test1();
  
```

Figure 2: Illustrative Example - CVE-2019-0609.

exceed that initially computed boundaries of the object, overwriting then the function's stack. This causes a memory corruption that could be leveraged for remote command execution.

This vulnerability is classified as an Out-Of-Bounds (OOB) by the CWE classification (CWE-787). The Common Weakness Enumeration (CWE) is a standard classification for software vulnerabilities, including description, scores, and technical information, widely adopted by practitioners and researchers (MITRE, 2023). The vulnerability triggered by the code in Figure 2 is fixed with a commit that patches two files: `EngineInterfaceObject.cpp` and `JavascriptLibrary.cpp`.

The patch adds new functionality to existing functions; both files are part of the compiler module. By collecting data from security-related commits similar to this one, we can characterize the modified files using metrics, count and group files more related to security, evaluate the main modules affected by security-related commits, and identify the potential vulnerability types that are more associated with JavaScript engines.

3 STUDY SETTING

In order to characterize potential vulnerable code of JavaScript engines by using information extracted from security-related commits, we formulated the following research questions (RQs):

RQ₁: What Are the Differences Between Security-Related Commits to Other Commits in JavaScript Engines? This question aims to characterize secu-

rity related commits in comparison with other commits from a software metric perspective; to do so, we adopt well-known metrics used in other bug and vulnerability prediction studies.

RQ₂: Which Are the Files and Modules in JavaScript Engines Most Modified by Security-Related Commits? This question identifies the most modified files by security-related commits; this may indicate security critical parts of the JavaScript engine. So, this information may guide further security analysis on the software and flag components where the security problems may happen in the future. Besides the files, this question also maps, when applicable, to the modules discussed in Section 2 (namely, parser, interpreter, compiler, and optimizer).

RQ₃ What Are the Vulnerability Types Potentially Connected to Security-Related Commits of JavaScript Engines? This question analyzes a subset of security-related commits and their potential connection to known vulnerability types.

Responses for these RQs give insights about the JavaScript engines and their security flaws, helping to define what is expected to be identified on this type of software. The findings can also support developers to quantify in which types of vulnerability the development effort for security (commits) has been spent or still instruct researchers while assessing the engines.

Analyzed Projects: To gain access to all pieces of information needed to answer the RQs, we first opted for JavaScript engines that are open source and available in GitHub. Second, we prioritized engines that are embedded in popular Web browsers. Finally, we also considered the domain and balanced different projects age. The 4 engines selected are briefly described as follows.

V8 is a JavaScript engine developed by Google in the context of the Chromium project. Initially released in 2008, it powers the current market leader Google Chrome, as well as other Chromium browsers like Microsoft Edge, Opera, and Samsung Internet. V8 is a mature project that also has contributors from different corporations, beside Google. V8 has also been adopted in popular server-side runtimes like Node.js and Deno.js. JavaScriptCore is the JavaScript engine of Apple’s Web browser framework WebKit. Starting in 2001, it powers several Apple software products like MacOS and iOS Safari browser. ChakraCore is a fork of the proprietary engine developed and used in Internet Explorer since 2009. In 2015, it was made open source by Microsoft, although newer versions of Edge use V8, ChakraCore is still maintained and remains a community project. Hermes is a JavaScript engine, developed by Face-

book, optimized for mobile devices; it was released in 2019. Hermes helps to reduce start-up time and decrease memory usage in mobile apps developed using the cross-platform framework React-Native.

Table 1 gives an overview of the 4 JavaScript engines projects; it shows the number of commits, number of unique contributors (#Devs), main sponsor and uses, lines of code (LoC), number of files, number of classes, and number of methods. Concerning size in LoC, V8 is the biggest project: 68,036 commits, more than 1,900 KLoC, and also the project with more C/C++ files, as well as classes and methods.

Next, it is ChakraCore with more than 1,012 KLoC, 1,377 classes and 33,373 methods. While JavaScriptCore is the 3rd project in size (around 673 KLoC), it has more commits than ChakraCore and more developers involved in the project than in V8.

Hermes is the smallest and youngest project, with 517.6 KLoC and 2,602 commits. It also has only contributors from Facebook.

Commit Selection. To answer the posed questions, we first need to propose a reliable mean to retrieve security-related commits from the projects. Differently from previous work (Chang et al., 2011; Neuhaus and Zimmermann, 2010), we aim to go beyond known CVE vulnerabilities and analyze commits that handle any aspect of security. For this end, this study intends to include refactoring, minor bug fixes, and new features that are in some way related to security. Clearly, major vulnerabilities (with CVE or not) are also taken into account. As in other works (Barnett et al., 2016; Wang et al., 2021), we leverage the message sent along with a Git commit to identify and classify security-related commits. As the selected projects are maintained by major software companies, the contributions are mostly performed by their employees and follow best practices like accurate descriptions of commits’ messages. To collect Git information, we adopted PyDriller (Spadini et al., 2018).

To scale the selection of security-related commits, we developed a classifier that uses the commit message to identify whether it is security-related or not. First, we manually inspected and classified a dataset of 200 commit messages: half were related to some aspect of security, and the other half were not related.

The messages were retrieved from the Git repositories of the 4 selected JavaScript engines. The messages were randomly chosen and then analyzed until finding 25 security-related commits and 25 not related (others), for each engine.

The classification of commit messages was performed by the first author, who is a software security

Table 1: Overview of the JavaScript engines projects.

Engine	#Commits	#Devs	Sponsor	Main uses	LoC	#Source Files	#Classes	#Methods
ChakraCore	12,929	281	Microsoft	IE/Edge	1,012,041	1,853	1,377	33,373
Hermes	2,602	219	Facebook	React Native	517,614	1,399	2,475	22,289
JavaScriptCore	18,747	1,597	Apple	Safari, iOS	673,558	2,788	841	15,916
V8	68,036	911	Google	Chrome, Node.js	1,900,848	3,033	4,398	84,229

professional with more than 12 years of experience. The classification was also reviewed by the other authors with experience on mining software repositories; inconsistencies were discussed in sync meetings.

During the manual analysis of the messages, we selected keywords that may characterize a security aspect of commit. To do so, we extracted characteristics using *Bag-of-Words* (Zhang et al., 2010) to break the messages into a vector for further analysis. From the vectored messages, stop-words were removed and the most utilized words related to security were identified. We also extended this set of keywords with expert knowledge of the researchers involved.

The following keywords and their frequency were used as features to train the classifier: access, auth, bypass, confus, CVE, CWE, danger, denial of service, disclosure, ensure, exception, exploit, failure, harmful, incorrect, issue, leak, malicious, null, overflow, pass, password, prevent, safe, secur, sensitive, state, unauth, uninitialized, user-after-free, vulnerab.

We opted to train a classifier because a simple keyword search would not detect all relevant commits. We used the Python library scikit-learn (Pedregosa et al., 2011) to train a classifier using the keywords and the dataset of 200 commit messages labeled manually. After the training with six different algorithms and using 10-fold cross-validation, we selected the best classifier, which adopts Perceptron and achieved 84.5% accuracy and 84.2% F1-score.

As we are interested in the source code of the engines, we filtered out modifications to files that are not C/C++ code (namely, file extensions .c, .cc, .cpp and .h). We also removed commits without changes to source code. The classifier indicated 4,482 commits as security-related from a total of 102,314 commits analyzed. These security-related commits are distributed as shown in Table 2. We then randomly selected the same number of commits, per engine, classified as other (no security-related) for our analysis.

Table 2: Classification Results.

Engine	#Security-related commits
V8	3,407
ChakraCore	649
JavaScriptCore	233
Hermes	193

Metrics Collection. To answer **RQ₁**, we adopted code metrics to characterize and distinguish the security-related commits from other commits. The first metrics are related to modified lines and methods; we used PyDriller (Spadini et al., 2018) to collect the following metrics: i) lines added; ii) lines removed; iii) lines added + removed; iv) diff methods: the difference of added and removed methods; and v) changed methods. We also used the Understand tool, adopted in other security-related papers (Zhou et al., 2021; Medeiros et al., 2017; Shin et al., 2011). We collected a total of 39 metrics. More details about these metrics are found in our experimental package.

Thus for each commit, we selected the metrics' values for each file changed, and then mean values were calculated. To make comparisons, we first performed the Mann-Whitney U, a non-parametric test, to identify the statistical difference between the security-related and other commits, for each metric. For a significance level of 0.05 (p -value < 0.05), the result indicates that there is a statistical difference between them. To measure the effect size, we used Cliff's Δ , also recommended for non-parametric tests (Kitchenham et al., 2017). Using reference values from the literature (Kitchenham et al., 2017), values under 0.112 indicate a negligible effect; values between 0.112 and 0.276, a small effect; between 0.276 and 0.428 medium; and values greater than 0.428 indicate a large effect size.

For **RQ₂** we calculated how many modifications (frequency) were made in source code files. We also mapped each of the Top-10 most modified files, per engine, to the main modules composing a JavaScript engine (see Section 2). This mapping was done by first checking the documentation in the file's header, and the folder structure of the file. We also observed relevant comments in other parts of the file. Finally, we inspected the source code to determine which module the file was linked to. We skipped files that are related to different modules like tests.

Concerning **RQ₃**, we randomly selected 5% of the security-related commits and manually classified 355 security-related commits, 65 from ChakraCore, 50 from JavaScriptCore, 190 from V8 and 50 from Hermes, with respect to potential vulnerability types. As 5% of the commits for JavaScriptCore and Hermes correspond to few commits, we decided to select a larger sample with 50 commits for them.

The types of vulnerabilities were defined while inspecting the commits, using as basis the CWE. For the classification, we first observed the commit title and messages, searching for keywords that could indicate the type of vulnerability. Then, we checked external references as CVE IDs if available, which can also disclose technical information about the vulnerability. Finally, we tried to understand the issue treated in the messages for the classification. We skipped commits with lacking information. CWE has multiple classifications for types of vulnerabilities; for example, Out-Of-Bounds (OOB) can be described as OOB Read (CWE-125) and OOB Write (CWE-787), so we grouped these related categories. We also grouped vulnerabilities with parent-child relations, such as Generic Crash (CWE-248), Generic Leak (CWE-200), Generic Overflow (CWE-119), and Arbitrary Privileges (CWE-269). We classified the security-related commits using the entire CWE database and ultimately, 9 types of vulnerability were observed: Arbitrary Privileges, Generic Crash, Generic Leak, Generic Overflow, Out-Of-Bounds (OOB), Race Condition, Use-After-Free (UAF), Type Confusion, and Null Pointer Dereference.

4 ANALYSIS OF RESULTS

In this section we analyze the information collected from the selected commits to answer the RQs.

4.1 RQ₁: Security-Related Commits and Others

Table 3 shows for each metric and engine, whether security-related commits are different from other commits (p -value), and how meaningful is the difference via effect size (Cliff's Δ). For a given metric in a row, symbol + represents a statistically significant difference (p -value < 0.05), otherwise symbol - is used (p -value ≥ 0.05). Observe that no metric has a large or medium effect size in any engine, so all effect size values are below 0.276. A small effect is represented with the symbol * and no mark is used for a negligible effect size.

In ChakraCore, security-related commits are distinguished from other commits: there is statistical difference for 42 metrics (out of 44). For 38 metrics the effect size is small, and for only 6 is negligible. JavaScriptCore and Hermes show intermediate results. In JavaScriptCore there is statistical difference for 30 metrics; for 29 the effect size is small, and for 15 is negligible; while for Hermes there is statistical difference for 23 metrics, for 20 metrics the effect

size is small, and for only one the effect is negligible. In V8, the differences are subtle: only for 16 out of 44 metrics, there is a significant difference, and all effect sizes are negligible.

There is statistical difference for only six metrics (around 14% of them) when all the four projects are considered. These metrics are highlighted in red in Table 3 and characterize some complexity aspects of the code. For other 22 metrics there is statistical difference in 3 projects: most of these metrics are related to counting lines or statements.

There is difference for five metrics in only one project, and for one metric (lines.added+removed, highlighted in blue) there is no statistical difference in any project.

For 17 metrics the effect size is small for 3 projects (7 metrics are complexity-related and 10 count code structures). For other five metrics (4 from PyDriller) the effect size is negligible in all projects. The top 5 highest effect size values are for metrics in ChakraCore: 0.245 for MaxNesting, 0.243 for SumCyclomaticModified, 0.242 for SumCyclomaticStrict, 0.239 for SumEssential, and 0.236 SumCyclomatic. On the other hand, the lowest values of effect size are for V8 and Hermes.

Figure 3 shows boxplots for (a) MaxEssential and (b) lines.added+removed. MaxEssential is one of the 6 metrics that are statically different for all projects. Notice that security-related commits and others are pretty similar for V8 and Hermes (negligible effect), while there are more differences in JavaScriptCore and ChakraCore (higher values of effect size). For lines.added+removed, we observe more similarity (no statistically difference and negligible effect size).

Response to RQ1: When the 4 JavaScript engines are taken into account, there is statistical difference between security-related commits and others commits in the values of only six metrics: AvgCyclomaticModified, AvgEssential, MaxEssential, SumCyclomatic, SumCyclomaticModified, and SumCyclomaticStrict. The effect sizes are most of the times small. No major pattern was observed for all engines; for JavaScriptCore and ChakraCore engines, the metrics seem to distinguish security-related commits, while for V8 does not.

Implications: Complexity metrics demonstrated a reasonable difference, even with a small effect size. The difference indicates a relationship between complexity metrics and software security. Similar results were also found in the literature with another JavaScript engine, and complexity metrics seem to

Table 3: Metrics Statistics and Effect Size (RQ₁).

Metric	ChakraCore		Hermes		JavaScriptCore		V8	
	p-value	Cliff's Δ	p-value	Cliff's Δ	p-value	Cliff's Δ	p-value	Cliff's Δ
lines_added	0.0397022 (+)	0.070	0.4109654 (-)	0.013	0.5696935 (-)	0.030	0.0029788 (+)	0.042
lines_removed	0.0283019 (+)	0.074	0.4853447 (-)	0.002	0.6036602 (-)	0.027	0.5924578 (-)	0.008
lines_added+removed	0.5288805 (-)	0.021	0.4382905 (-)	0.009	0.5302839 (-)	0.033	0.0693821 (-)	0.026
diff_methods	0.4209284 (-)	0.022	0.0654076 (-)	0.078	0.3749170 (-)	0.042	0.0265228 (+)	0.029
changed_methods	0.0000325 (+)	0.137 *	0.4611209 (-)	0.006	0.0720051 (-)	0.093	0.0000000 (+)	0.092
AltAvgLineBlank	0.0000007 (+)	0.168 *	0.2571563 (-)	0.033	0.0003598 (+)	0.176 *	0.0010343 (+)	0.047
AltAvgLineCode	0.0000000 (+)	0.197 *	0.0232725 (+)	0.112 *	0.0003807 (+)	0.183 *	0.1188857 (-)	0.023
AltAvgLineComment	0.0000008 (+)	0.182 *	0.4054388 (-)	0.009	0.0190639 (+)	0.109	0.7179380 (-)	0.004
AltCountLineBlank	0.0000002 (+)	0.192 *	0.1087684 (-)	0.068	0.2519423 (-)	0.060	0.0237610 (+)	0.031
AltCountLineComment	0.0001801 (+)	0.127 *	0.1160282 (-)	0.065	0.1767363 (-)	0.070	0.0010902 (+)	0.045
AvgCyclomatic	0.0000001 (+)	0.213 *	0.0357629 (+)	0.101	0.0008727 (+)	0.171 *	0.1178244 (-)	0.023
AvgCyclomaticModified	0.0000001 (+)	0.221 *	0.0228974 (+)	0.112 *	0.0009687 (+)	0.170 *	0.0011943 (+)	0.047
AvgCyclomaticStrict	0.0000000 (+)	0.224 *	0.0585249 (-)	0.087	0.0007105 (+)	0.174	0.2543875 (-)	0.017
AvgEssential	0.0000002 (+)	0.189 *	0.0060227 (+)	0.139 *	0.0012841 (+)	0.163 *	0.0091896 (+)	0.037
AvgLine	0.0000002 (+)	0.195 *	0.0472579 (+)	0.094	0.0002501 (+)	0.189 *	0.1424444 (-)	0.022
AvgLineBlank	0.0000003 (+)	0.174 *	0.2609203 (-)	0.032	0.0003401 (+)	0.176 *	0.0024615 (+)	0.044
AvgLineCode	0.0000001 (+)	0.209 *	0.0174867 (+)	0.119 *	0.0002139 (+)	0.191 *	0.1698936 (-)	0.021
AvgLineComment	0.0000003 (+)	0.190 *	0.3514346 (-)	0.017	0.0061876 (+)	0.124 *	0.6741754 (-)	0.005
CountDeclClass	0.0013618 (+)	0.104	0.3910023 (-)	0.021	0.0665574 (-)	0.090	0.0033923 (+)	0.043
CountLine	0.0000002 (+)	0.193 *	0.0203062 (+)	0.116 *	0.3304187 (-)	0.051	0.0258001 (+)	0.030
CountLineBlank	0.0000000 (+)	0.204 *	0.0865085 (-)	0.075	0.0602482 (-)	0.098	0.9704849 (-)	0.002
CountLineComment	0.0002615 (+)	0.124 *	0.1128739 (-)	0.066	0.2500900 (-)	0.060	0.1964651 (-)	0.017
CountLineInactive	0.0012380 (+)	0.110	0.1949235 (-)	0.056	0.0015683 (+)	0.164 *	0.0046214 (+)	0.039
CountSemicolon	0.0000000 (+)	0.215 *	0.0079426 (+)	0.137 *	0.0000623 (+)	0.207 *	0.1354262 (-)	0.022
CountStmt	0.0000000 (+)	0.220 *	0.0078057 (+)	0.138 *	0.0000582 (+)	0.208 *	0.1404430 (-)	0.022
CountStmtDecl	0.0000000 (+)	0.213 *	0.0061338 (+)	0.143 *	0.0000884 (+)	0.202 *	0.1250137 (-)	0.023
CountStmtEmpty	0.0000000 (+)	0.222 *	0.2410181 (-)	0.036	0.0072206 (+)	0.115 *	0.0000000 (+)	0.094
CountStmtExe	0.0000000 (+)	0.221 *	0.0096755 (+)	0.133 *	0.0000679 (+)	0.205 *	0.0631482 (-)	0.028
MaxCyclomatic	0.0000000 (+)	0.202 *	0.0915054 (-)	0.073	0.0002143 (+)	0.191 *	0.0075584 (+)	0.039
MaxCyclomaticModified	0.0000000 (+)	0.234 *	0.0784226 (-)	0.078	0.0001546 (+)	0.195 *	0.0005768 (+)	0.050
MaxEssential	0.0000001 (+)	0.194 *	0.0294790 (+)	0.106	0.0000961 (+)	0.201 *	0.0210106 (+)	0.034
RatioCommentToCode	0.0003504 (+)	0.122 *	0.0266136 (+)	0.119 *	0.0587783 (-)	0.098	0.0612620 (-)	0.025
SumCyclomatic	0.0000000 (+)	0.236 *	0.0047370 (+)	0.148 *	0.0001487 (+)	0.195 *	0.0419466 (+)	0.030
SumCyclomaticModified	0.0000000 (+)	0.243 *	0.0037159 (+)	0.153 *	0.0001438 (+)	0.196 *	0.0151174 (+)	0.036
AltCountLineCode	0.0000000 (+)	0.200 *	0.0110735 (+)	0.130 *	0.3856598 (-)	0.045	0.0580866 (-)	0.026
CountLineCodeDecl	0.0000019 (+)	0.162 *	0.0022665 (+)	0.162 *	0.0001502 (+)	0.196 *	0.1233321 (-)	0.023
CountDeclFunction	0.0000000 (+)	0.216 *	0.0043810 (+)	0.150 *	0.0003753 (+)	0.183 *	0.1004145 (-)	0.025
CountLinePreprocessor	0.0000716 (+)	0.135 *	0.4422296 (-)	0.014	0.0251957 (+)	0.117 *	0.4300351 (-)	0.010
CountLineCode	0.0000000 (+)	0.194 *	0.0061900 (+)	0.142 *	0.0000690 (+)	0.205 *	0.3708459 (-)	0.014
CountLineCodeExe	0.0000000 (+)	0.185 *	0.0019733 (+)	0.165 *	0.0000473 (+)	0.210 *	0.1910002 (-)	0.020
MaxCyclomaticStrict	0.0000000 (+)	0.209 *	0.1103473 (-)	0.067	0.0001969 (+)	0.192 *	0.0110004 (+)	0.037
SumCyclomaticStrict	0.0000000 (+)	0.242 *	0.0054549 (+)	0.145 *	0.0001287 (+)	0.197 *	0.0404654 (+)	0.030
SumEssential	0.0000000 (+)	0.239 *	0.0019284 (+)	0.165 *	0.0002034 (+)	0.191 *	0.1094146 (-)	0.024
MaxNesting	0.0000000 (+)	0.245 *	0.0075804 (+)	0.138 *	0.0027461 (+)	0.154 *	0.0752023 (-)	0.027

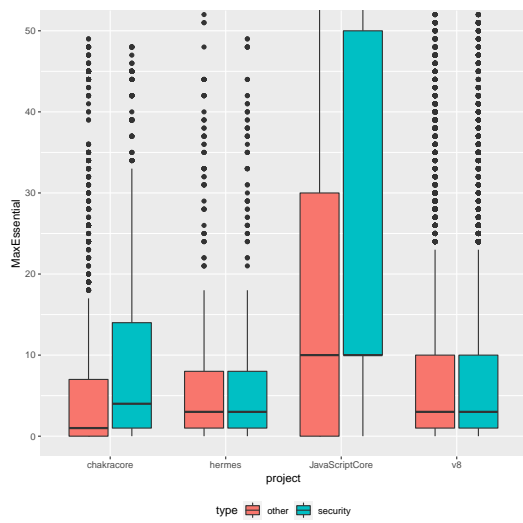
be effective for vulnerability prediction (Shin and Williams, 2008). Source files with high complexity metrics are common targets for security professionals to pursue vulnerabilities and significantly improve static and dynamic analysis results (Shin and Williams, 2011). This fact seems valid for JavaScript engines, though other metrics are significantly different only for some engines. Based on the observed effect size, it is advisable to complement metrics' values (as the ones shown in this RQ) with other features when analyzing the security of JavaScript engines.

4.2 RQ₂: Files and Modules Most Modified

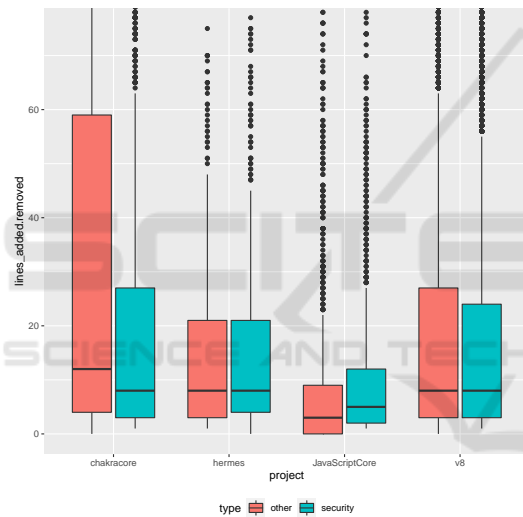
To answer RQ₂ we generated Figure 4. This figure shows the top 10 files that are most modified in the

security-related commits we analyzed. We also generated a rank of the top 10 files modified by other commits. Files that belong to both ranks are highlighted in purple in the same figure.

In ChakraCore (Figure 4a), the most modified file is GlobOpt.cpp (modified by 100 commits, around 16%). This file implements multiple optimization techniques, such as Optimize, OptLoops, and ForwardPass. When looking at top 10 most modified files in other commits (not security-related), only one file (Lower.cpp – also related to optimizations) appears in both types of commits we analyzed. Overall, 860 files are modified by security-related commits and 1,462 modified by other commits; there is an intersection of 622 files (~72%) modified by both types of commits, and 238 (27%) files modified only by the security-related ones.



(a) MaxEssential metric.

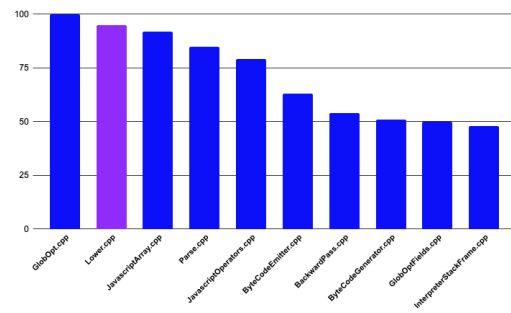


(b) Line changes metric.

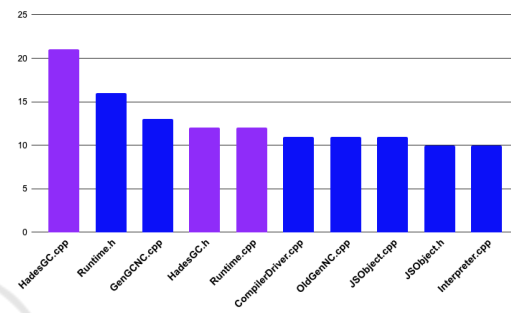
Figure 3: Comparing security-related commits and others.

In Hermes (Figure 4b), the most modified file is HadesGC.cpp (modified by 21 commits, ~10%). This file is related to the garbage collector, cleaning the memory from unnecessary allocations and speeding up the execution. There are 3 files that are in the top-10 most modified of both types of commits; an example is Runtime.cpp, a file related to the compiler. Overall, 394 files are modified by security-related commits and 271 files modified by other commits; there is an intersection of 177 (~44%) files modified by both types of commits, and 217 (~55%) files modified only by the security-related ones.

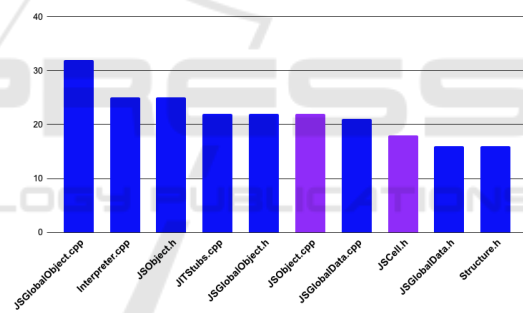
In JavaScriptCore (Figure 4c), the most modified file is JSGlobalObject.cpp (by 32 commits, around 13%). This file handles the objects created during



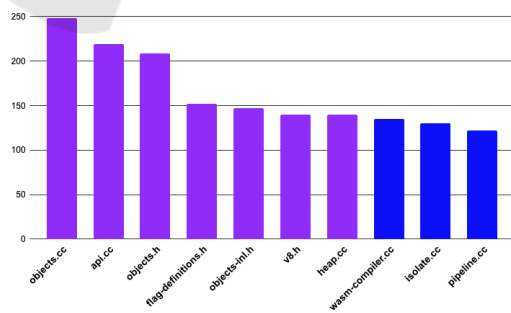
(a) ChakraCore.



(b) Hermes.



(c) JavaScriptCore.



(d) V8.

Figure 4: Top-10 most modified files by security commits and their intersection with other commits.

the execution. There are 2 files that are in the top-10 most modified of both types of commits: JSObject.cpp that manipulate JavaScript objects and a JSCell.h, a header file utilized by the compiler. Over-

all, 372 files are modified by security-related commits and 1081 files modified by other commits. There is an intersection of 338 (~90%) files modified by both types of commits, and 34 (~10%) files modified only by the security-related ones.

In V8 (Figure 4d), the most modified file is `objects.cc` (modified by 32 commits (~9%)); this file is generic, and it has multiple functions imported from all modules. Because of that, the same file happens to be the most modified in other commits too. There are 7 files that are in the top-10 most modified of both types of commits: `objects.cc`, `api.cc`, `objects.h`, `flag-definitions.h`, `objects-inl.h`, `v8.h` and `heap.cc`. Overall, 1,616 files are modified by security-related commits and 3,033 files modified by other commits; there is an intersection of 1,341 (~81%) files modified by both types of commits, and 275 (~19%) files modified only by the security-related ones.

Security-related commits modified a set of files smaller than other commits, except for Hermes. While around half of the total files are modified by both types of examined commits, (around 55%, considering all engines), yet there is a reasonable number of files (~17%) that are changed only by security-related commits during this restricted analysis. This evinces that some files are more impacted by security aspects, and the results herein presented may be used to pinpoint them.

Figure 5 shows the modules modified by the top-10 files in security-related commits, per engine. Observe that security-related commits interfere mostly in the optimizer in the ChakraCore, JavaScriptCore, and Hermes. In V8, the most modified type of module is the compiler, followed by the optimizer. The second most modified type is the compiler. The interpreter comes in third and the parser appears in ChakraCore and JavaScriptCore.

Response to RQ2: We identified the top 10 most modified files by security-related commits. Most of them (~67%) are not in the top 10 most modified files of the other commits. While there exists an intersection of files modified by both types of commits, ~33% (13) of Top-10 files, most of them are changed only by security-related commits. The most-modified files in Hermes, ChakraCore, and JavaScriptCore relate directly to the optimizer, while in V8, the most-modified file relates to the compiler. Overall, the modules optimizer and compiler are the main focus of security-related commits, while the types parser and interpreter are less changed.

Implications: The information about files modified by security-related commits is essential for further research. Most of state-of-the-art fuzzing tools for JavaScript engines (Han et al., 2019; Lin et al., 2019; Holler et al., 2012) utilize PoC (Proof-of-Concept) files from existing vulnerabilities as seeds for input generation, which means that new security problems often come from known issues affecting the same file in different versions (Lee et al., 2020).

Our paper extends the existing results in the literature (Lee et al., 2020) and includes security-related commits that go beyond the ones that are CVE-related. This information could be used to boost the selection of files and modules targeted in a security assessment. We also extended the results, pinpointing which are the most affected modules by security-related commits. We can utilize this data and ensure that the top files and modules modified by security-related commits have enough attention or need a deeper inspection from a security perspective.

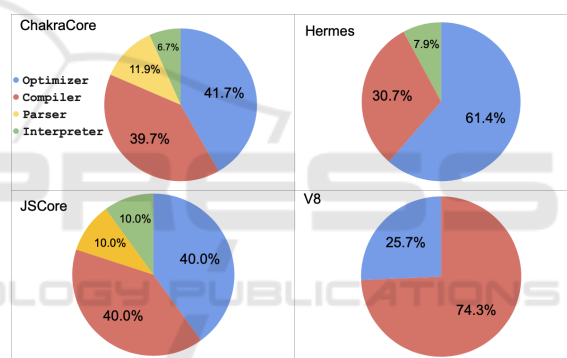


Figure 5: Modules modified by security-related commits.

4.3 RQ₃: Vulnerability Types in Security-Related Commits

To answer RQ₃ we refer to Figure 6, which shows, per engine, the distribution of vulnerability types that may be related to the security-related commits we analyzed. Some vulnerabilities are more prevalent: in ChakraCore, the Top-3 types that appear more are: Out-Of-Bounds, Generic Crash, and Type Confusion; in Hermes, they are: Generic Leak, Type Confusion, and Generic Crash; in V8, they are: Generic Crash, Generic Leak, and Type Confusion; and in JavaScriptCore, they are: Generic Crash, Type Confusion and Out-Of-Bounds. Notice that Generic Crash and Type Confusion are prevalent in all engines; the types Generic Leak and Out-Of-Bounds appear in 2 engines.

By looking at the most prevalent types associated

to security-related commits, we observe that all engines may be subjected mostly to specific vulnerabilities. Generic Crash is concerned to whenever the engine can not handle an exception, it stops the execution. Type Confusion is a security issue frequently exploited in JavaScript engines (Sun et al., 2022); it typically explores the optimization feature by forcing the engine to identify a variable as a specific type, and then a function handles the variable as another type, causing the vulnerability that can be leveraged to compromise memory data.

Generic Leak and Out-Of-Bounds are also prevalent. Generic Leak occurs when any information is disclosed arbitrarily; for instance, relevant system addresses could be leaked and then utilized for bypassing memory randomization protection.

Out-Of-Bounds happens when an index refers to a memory location outside of the buffer’s boundaries; for example, when there is an array and an arbitrary index is provided, the program would return or overwrite arbitrary data. Considering all engines, Use-After-Free (UAF), Arbitrary Privileges, Race Condition are the types of vulnerabilities that are least frequent. Generic Overflow and Null Pointer Dereference do not occur a lot too (less than 10%).

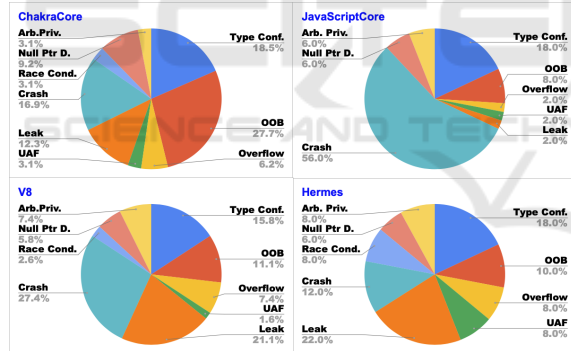


Figure 6: Types of vulnerabilities identified.

Response to RQ3: Security-related commits in JavaScript engines are related to 9 different types of vulnerability. Generic Crash and Type Confusion are in the Top-3 potential vulnerabilities, found in all engines. Generic Leak and Out-Of-Bounds are prevalent in 2 engines. Other vulnerabilities occur in few commits, such as Use-After-Free, Arbitrary Privileges, Race Condition, Generic Overflow, and Null Pointer Dereference.

Implications: We may draw a vulnerability portrait about which vulnerabilities often occur in JavaScript engines and provide insights to define methods

for their identification. For example, the Kop-Fuzzer (Sun et al., 2022) took advantage of this kind of specific knowledge to define a technique focused on Type Confusion. Such a type of security issue is prevalent in JavaScript engines; the authors reported the identification of at least 21 bugs in ChakraCore and JavaScriptCore. Knowing the vulnerability types associated to security-related commits would help security professionals to establish procedures for uncovering vulnerabilities, as well as determine patterns to detect them with tools like CodeQL.

Along with the findings of RQ₂, we can build (i) a reduced attack surface for researchers where a specific vulnerability type is targeted (e.g., by defining specific oracles); and (ii) an opportunity for developers to review the common development mistakes that lead to these security bugs in the JavaScript engines.

5 THREATS TO VALIDITY

This study is subjected to some threats we discuss next. The study did not consider all existing JavaScript engines, so the results may not generalize. To mitigate this, we selected a diverse set of engines, varying their browsers, main use cases, and project age. To define the training set, we manually classified the commit messages into security-related or not. While the classification was reviewed, some commits may be misclassified. Some commits may be wrongly flagged by our classifier, besides its high values of accuracy and F1-score. To assess the need for further training iterations, we manually checked random samples of 100 commits per engine labeled by the classifier. We found an accuracy of 85%.

As for RQ₂ and RQ₃, we performed steps that involved the manual classification of modules (associating files to engine modules) and potential relation to vulnerability types. The number of analyzed commits for distinguishing the type vulnerability is small, but they are representative and provided preliminary results. Besides, several steps of our analyses were automated by scripts and existing tools. Hence, the results may be impacted by flaws in the classification and implementation. We cross-checked the results and constantly discussed about them in order to mitigate those potential threats.

6 RELATED WORK

Several studies investigate the relation between security and software metrics (Zaman et al., 2011; Alves et al., 2016; Iannone et al., 2022). Most of them

have as focus to understand vulnerabilities, by using metrics to predict them or to pinpoint where they are (Shin and Williams, 2008; Shin et al., 2011; Jimenez et al., 2019). To this end, a first step is to collect data about security and vulnerabilities. To obtain pieces of information about vulnerabilities, popular sources like CVEs and NVD are used to search for known software vulnerabilities. There are also repositories like Exploit-DB and Metasploit. Moreover, there are some initiatives from academia. For instance, Gkortzis et al. (Gkortzis et al., 2018) put together a dataset of 8,694 open source projects with known vulnerabilities from NVD, along with essential metrics like the number of files and lines of code. Kiss and Hodován (Kiss and Hodován, 2019) developed a tool to acquire security-related commits from open source web browser projects. They leveraged security-labeled data in the GitHub and Bugzilla repositories to track issues and commits.

Other studies follow similar approaches to obtain a dataset for analyses. Zaman et al. (Zaman et al., 2011) analyze bugs related to the security and performance for the Firefox project. The authors observed that security bugs are organized and fixed much faster, involve more developers and usually impact more files in the project. Alves et al. (Alves et al., 2016) analyzed 2,875 security patches from 5 open-source projects. They observed differences in functions' metrics between the vulnerable functions and non-vulnerable functions. Iannone et al. (Iannone et al., 2022) analyzed 3,663 vulnerabilities from 1,096 GitHub projects to determine how developers introduce vulnerabilities to the software.

A popular line of research is to adopt software metrics and other pieces of information to predict vulnerabilities. Shin et al. (Shin and Williams, 2008) investigate two metrics: the modified cyclomatic complexity and strict cyclomatic complexity for which determined that vulnerable functions have distinctive characteristics from non-vulnerable functions. The authors validated the hypothesis that functions identified as more complex, with more loops and conditionals, indicated vulnerabilities in the future. In an extended study (Shin et al., 2011), the authors worked on how metrics can predict vulnerabilities. The prediction models were validated using the software releases and data sources such as Bugzilla, NVD, and the Red Hat Security Advisory. They could predict 70.8% of the known vulnerabilities in Firefox and 68.8% in the Linux Kernel.

Existing studies on vulnerability prediction have been criticized; Jimenez et al. (Jimenez et al., 2019) argue that they do not provide realistic results since they take in consideration only the "ideal" world for

prediction. Using three state-of-the-art prediction approaches, the authors observed a meaningful drop of predictive effectiveness when employing a more realistic scenario. Another explored direction is to avoid relying on historical data. For instance, Du et al. (Du et al., 2019) developed a framework named LEOPARD that identifies vulnerabilities in source code. The tool relies on a combination of heuristics and code metrics for C/C++ programs, being able to uncover multiple memory corruption vulnerabilities.

While the aforementioned works seek to investigate their questions in security-critical applications, none focuses on JavaScript engines and their characteristics, which is the focus of this paper. Moreover, we went beyond the existing labeled data from NVD and other repositories by selecting any security-related commits. The results herein presented may support future studies in security of JavaScript engines and contribute to this emergent research topic.

7 CONCLUDING REMARKS

This paper characterizes security-related commits of four widely used JavaScript engines. We analyzed code metrics, the most modified files and components, as well as the potential types of vulnerabilities associated. The obtained results show statistical difference between the security-related commits and others for mainly code complexity-related metrics. We observed that the modules most-affected by security-related commits are the optimizer and compiler. Nine different vulnerability types were associated with the security-related commits; the most prevalent are Generic Crash, Generic Leak, Type Confusion, and Out-of-Bounds.

Future replications may be conducted with more JavaScript engines and analyzing other security aspects. The presented results can be leveraged in future; the direct application would be to employ pieces of information about security-related commits for vulnerability prediction in JavaScript engines. The results can also determine a more focused attack surface for fuzzing tools, increasing the effectiveness and efficiency in uncovering new vulnerabilities.

ACKNOWLEDGEMENTS

This work is partially supported by CNPq (Andre T. Endo grant nr. 420363/2018-1 and Silvia Regina Vergilio grant nr. 305968/2018-1).

REFERENCES

- Alves, H., Fonseca, B., and Antunes, N. (2016). Software metrics and security vulnerabilities: Dataset and exploratory study. In *EDCC 2016*, pages 37–44.
- Barnett, J. G., Gathuru, C. K., Soldano, L. S., and McIntosh, S. (2016). The relationship between commit message detail and defect proneness in Java projects on GitHub. In *13th MSR*, pages 496–499.
- Chang, Y.-Y., Zavorsky, P., Ruhl, R., and Lindskog, D. (2011). Trend analysis of the CVE for software vulnerability management. In *IEEE PST*, pages 1290–1293.
- Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., and Jiang, Y. (2019). Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In *ICSE*, pages 60–71.
- Gkortzis, A., Mitropoulos, D., and Spinellis, D. (2018). Vulnoss: a dataset of security vulnerabilities in open-source systems. In *MSR*, pages 18–21. ACM.
- Han, H., Oh, D., and Cha, S. (2019). CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *NDSS*.
- Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with code fragments. In *USENIX*, pages 445–458.
- Iannone, E., Guadagni, R., Ferrucci, F., Lucia, A. D., and Palomba, F. (2022). The secret life of software vulnerabilities: A large-scale empirical study. *TSE*, (01):1–1.
- Jimenez, M., Rwemalika, R., Papadakis, M., Sarro, F., Traon, Y. L., and Harman, M. (2019). The importance of accounting for real-world labelling when predicting software vulnerabilities. In *ESEC/SIGSOFT FSE 2019*, pages 695–705. ACM.
- Kang, Z. (2021). A review on JavaScript engine vulnerability mining. *Journal of Physics: Conference Series*, 1744(4):042197.
- Kiss, A. and Hodován, R. (2019). Security-related commits in open source web browser projects. In *ASEW*, pages 57–60.
- Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., and Pohthong, A. (2017). Robust statistical methods for empirical software engineering. *Empir. Softw. Eng.*, 22(2).
- Lee, S., Han, H., Cha, S. K., and Son, S. (2020). Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *USENIX Security 20*, pages 2613–2630. USENIX Association.
- Lin, H., Zhu, J., Peng, J., and Zhu, D. (2019). Deity: Finding deep rooted bugs in JavaScript engines. In *2019 ICCT*, pages 1585–1594.
- Mao, J., Bian, J., Bai, G., Wang, R., Chen, Y., Xiao, Y., and Liang, Z. (2018). Detecting malicious behaviors in JavaScript applications. *IEEE Access*, 6:12284–12294.
- Medeiros, N., Ivaki, N., Costa, P., and Vieira, M. (2017). Software metrics as indicators of security vulnerabilities. In *ISSRE*, pages 216–227.
- MITRE (2023). Common Weakness Enumeration. <https://mitre.org/>.
- Neuhaus, S. and Zimmermann, T. (2010). Security trend analysis with CVE topic models. In *ISSRE*, pages 111–120.
- OffensiveCon (2022). Attacking JavaScript Engines in 2022. <https://www.offensivecon.org/speakers/2022/samuel-gro-and-amanda-burnett.html>.
- Park, S., Xu, W., Yun, I., Jang, D., and Kim, T. (2020). Fuzzing JavaScript engines with aspect-preserving mutation. In *IEEE S&P*, pages 1629–1642.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay (2011). Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.*, 12(85):2825–2830.
- Shin, Y., Meneely, A., Williams, L. A., and Osborne, J. A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *TSE*, 37(6):772–787.
- Shin, Y. and Williams, L. (2008). An empirical model to predict security vulnerabilities using code complexity metrics. In *ESME, ESEM '08*, page 315–317.
- Shin, Y. and Williams, L. (2011). An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *SESS*, page 1–7.
- Spadini, D., Aniche, M., and Bacchelli, A. (2018). Py-Driller: Python framework for mining software repositories. In *ESEC/FSE*, pages 908–911.
- SSLAB (2021). Analysis of a use-after-unmap vulnerability in Edge: CVE-2019-0609 - Systems Software and Security Lab. <https://gts3.org/2019/cve-2019-0609.html>.
- Sun, L., Wu, C., Wang, Z., Kang, Y., and Tang, B. (2022). Kop-fuzzer: A key-operation-based fuzzer for type confusion bugs in javascript engines. In *COMPSAC*, pages 757–766.
- US-Government (2023). National Vulnerability Database. <https://nvd.nist.gov/>.
- Wang, B., Yan, M., Liu, Z., Xu, L., Xia, X., Zhang, X., and Yang, D. (2021). Quality assurance for automated commit message generation. In *SANER*, pages 260–271.
- Zaman, S., Adams, B., and Hassan, A. E. (2011). Security versus performance bugs: a case study on Firefox. In *MSR*, pages 93–102. ACM.
- Zhang, Y., Jin, R., and Zhou, Z. (2010). Understanding bag-of-words model: a statistical framework. *Int. J. Mach. Learn. Cybern.*, 1:43–52.
- Zhou, A., Sultana, K. Z., and Samantha, B. K. (2021). Investigating the changes in software metrics after vulnerability is fixed. In *IEEE Big Data*, pages 5658–5663.